Implementing Joins

Last Time

- Selection
  - Scan, binary search, indexes
- Projection
  - Duplicate elimination: sorting, hashing
  - Index-only scans
- Joins

Tuple Nested Loop Join

```
foreach tuple r in R do
    foreach tuple s in S do
        if r.sid == s.sid  then add <r, s> to result
```

- R is "outer" relation
- S is "inner" relation

Page Nested Loop Join

```
foreach page p1 in R do
    foreach page p2 in S do
        foreach r in p1 do
            foreach s in p2 do
                if r.sid == s.sid  then add <r, s> to result
```

- R is “outer” relation
- S is “inner” relation

Analysis

- Assume
  - M pages in R, p_R tuples per page
    - $M = 1000$, $p_R = 100$
  - N pages in S, p_S tuples per page Select
    - $N = 500$, $p_S = 80$
- Total cost = $M + M * N$
  - Which is 501,000 I/O s

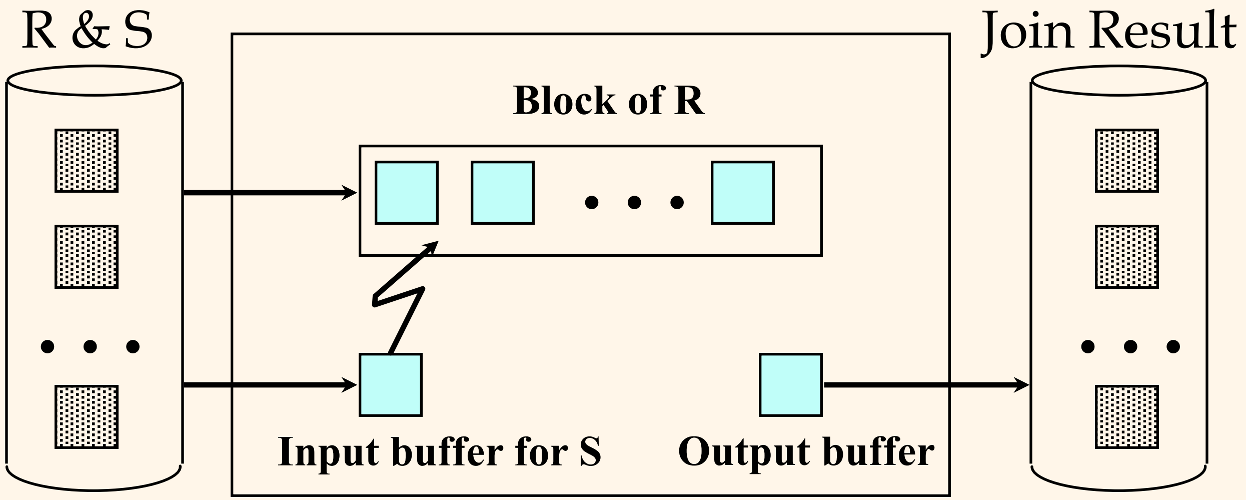
- Note: Smaller relation should be "outer"
  - Better for S to be "outer" in this case!

Block Nested Loop Join

- Notice: previous algorithms do not use all the available buffer pages
- We could keep all (or at least some) of the outer relation in memory while we scan the inner one

Block Nested Loops Join

- Use one page as input buffer for scanning S, one page as output buffer, and all remaining pages to hold ``block'' of R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil \# \ of \ pages \ of \ outer \ / \ blocksize \rceil$
- With Reserves (R) as outer, and 100 page blocks:
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.

Examples of Block Nested Loops

- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.

Index Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S where r_i == s_j do
        add <r, s> to result
```

- Suppose we have an index on S, on the join attribute
- No need to scan all of S – just use index to retrieve tuples that match this r
- This will probably be faster, especially if there are few matching tuples and the index is clustered

Index Nested Loops Join

foreach tuple r in R do
  foreach tuple s in S where $r_i == s_j$ do
    add <r, s> to result

- Cost: M + ((M*p_R) * cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: up to 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Join sailors and reserves on sid
- Hash index (Alt. 2) on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 221,000 I/Os.

Examples of Index Nested Loops

- Hash index (Alt. 2) on *sid* of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.
  - Final numbers: 88,500 (clustered), 148,500 (unclustered)

- Still not as good as the numbers we saw for BNLJ, but much better than TNLJ.

Sort-merge join motivation

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Sort-Merge Join

- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.

Sort-Merge Join on Ri = Sj

- Scan R and S until current R tuple = current S tuple.

- At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.

- Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple.
  - If the groups are huge the whole S-group may not fit in memory
  - Although this is rare in practice

Cost of Sort-Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Cost: (cost to sort R) + (cost to sort S) + (scan cost)
  - M < scan cost <= M+ M*N (typically close to M+N)
  - With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

(BNL cost: 2500 to 15000 I/Os)

Refinement of Sort-Merge Join

- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
- Let L be # pages of larger relation, S of smaller relation
- After pass 0, have L/B runs of larger, S/B runs of smaller
- Can merge them in one pass if total # runs <= B-1

Refinement of sort-merge join

- Want $$\frac{L}{B} + \frac{S}{B} \leq B - 1$$

- I.e. $$L + S \leq B^2 - B$$

- As a rule of thumb, need $B \geq 2\sqrt{L}$

SMJ cost with refinement

- 3 (L + S)
- Pass 0 for each relation contributes 2L and 2S
- Merge pass is L + S (typical case)

- In our example, 4500 I/Os

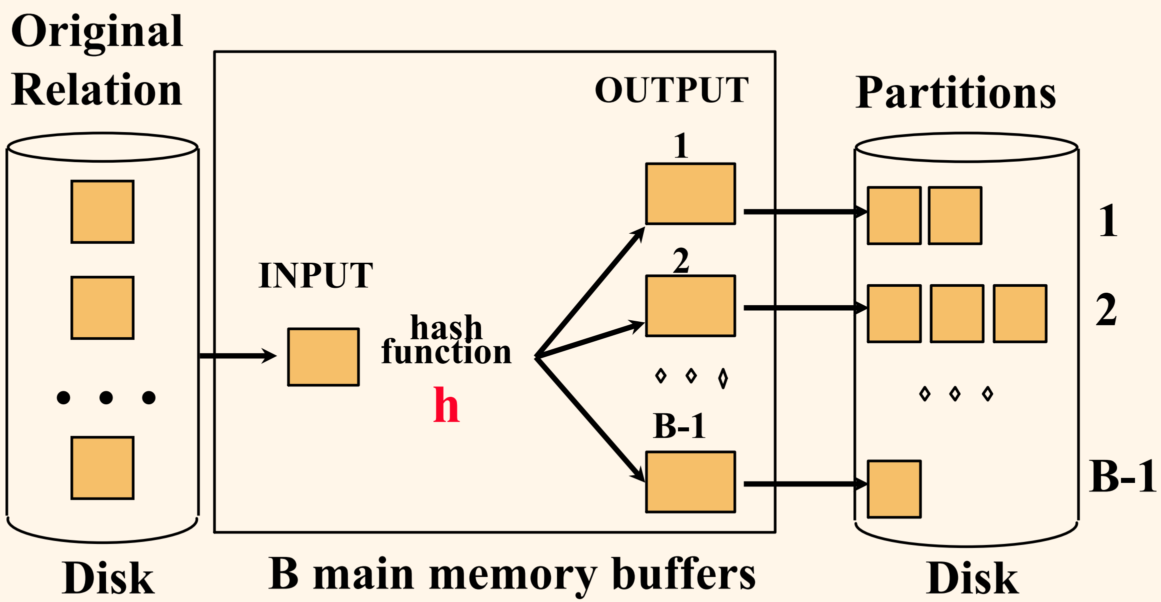
- In practice SMJ has linear cost (in I/Os)

Hash Join

- Similar idea to hashing for projection
- Hash both relations on join attribute
- Matching tuples fall into same partition
- Then can compute join on partitions one at a time

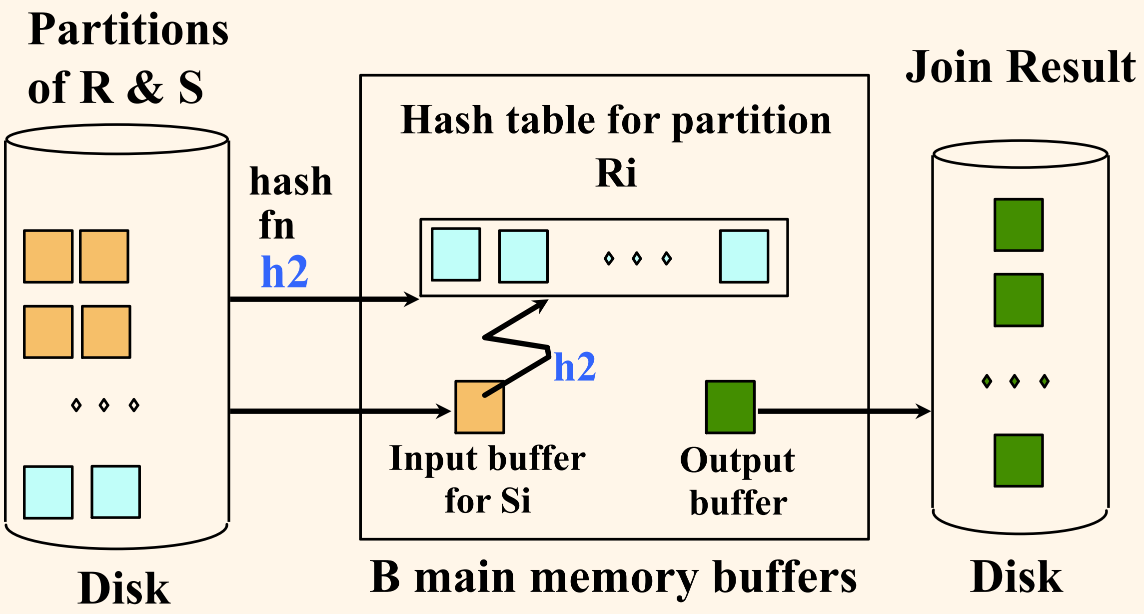
Hash Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.



Hash Join

- Read in a partition of R, hash it using **h2 (<> h!)**. Scan matching partition of S, search for matches.



Observations on Hash Join

- One or more R partitions may not fit in memory. If so, can apply hash join technique recursively:
  - Take partition of R and corresponding partition of S
  - Use different hash function to split each of them up into subpartitions
  - Join the subpartitions one at a time

Observations on Hash Join

- How much memory do we need so we don't have to recurse?
- #partitions k <= B-1, and B-2 >= size of largest partition to be held in memory. Assuming uniformly sized partitions, letting S = size of smaller relation, and maximizing k, we get:
  - k= B-1, and S/(B-1) <= B-2, i.e., B must be (approx) > $\sqrt{S}$
- Contrast with sort-merge join, where "enough buffer pages" was based on size of the *larger* relation

Cost of Hash Join

- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.
- In our running example, this is a total of 4500 I/Os.

Sort-Merge Join vs Hash Join

- Given a minimum amount of memory both have a cost of 3(M+N) I/Os.
- Hash Join superior on this count if relation sizes differ greatly (depends on size of smaller relation vs larger relation for sort-merge)
- Also, Hash Join is highly parallelizable.
- Sort-Merge Join less sensitive to data skew; also, result is sorted.